Construction of Strength Two Mixed Covering Arrays Using Greedy Mutation in Genetic Algorithm

Sangeeta Sabharwal
Netaji Subhas Institute of Technology, Sector-3, Dwarka, New Delhi-110078, India
E-mail: ssab63@gmail.com

Priti Bansal and Nitish Mittal
Netaji Subhas Institute of Technology, Sector-3, Dwarka, New Delhi-110078, India
E-mail: banalprit79@gmail.com, nitishmittal94@gmail.com

Abstract—Metaheuristic methods are capable of solving a wide range of combinatorial problems competently. Genetic algorithm (GA) is a metaheuristic search based optimization algorithm that can be used to generate optimal Covering Arrays (CAs) and Mixed Covering Arrays (MCAs) for pair-wise testing. Our focus in the work presented in this paper is on the strategies of performing mutation in GA to enhance the overall performance of GA in terms of solution quality and computational time (number of generations). This is achieved by applying a greedy approach to perform mutation at a position that minimizes the loss of existing distinct pairs in the parent CA/MCA and ensures that the generated offspring is of good quality. Experiments are conducted on several benchmark problems to evaluate the performance of the proposed greedy based GA with respect to the existing state-of-the-art algorithms. Our evaluation shows that the proposed algorithm outperforms its GA counterpart by generating better quality MCA in lesser number of generations. Also the proposed approach yields better/comparable results compared to the existing state-of-the-art algorithms for generating CAs and MCAs.

Index Terms—Pair-wise testing, mixed covering arrays, genetic algorithm, mutation, greedy approach.

I. INTRODUCTION

In a highly configurable software product, it is necessary to test the interaction among various configuration parameters to avoid interaction errors. For instance, in a system with n configuration parameters each of which can take m possible values, an exhaustive test set will have m^n test cases to check all possible combinations of configuration parameters. The number of test cases increases exponentially with the increase in number of configuration parameters. Thus, exhaustive testing of highly configurable software may be impractical due to the limitation of budget and time required to generate and run large sized test sets. An alternative to exhaustive testing is combinatorial interaction testing (CIT) as introduced in [1] which samples the set of configurations in such a way so as to test all possible t-way (t denotes the strength of testing) combinations of configuration parameters. The size of test set grows at most logarithmically in CIT with the increase in number of configuration parameters compared to the exponential growth in case of exhaustive testing [1].

Pair-wise testing is a CIT technique that tests all possible pair-wise (2-way) combinations of configuration parameter values. Pair-wise testing drastically reduces the size of test set as compared to exhaustive testing, without losing significantly on the fault detection capability [2]. Empirical studies show that test set covering all possible 2-way combination of configuration parameter values is effective for software systems [1, 3, 4]. In further work, Burr and Young [5] provided more empirical results to show that pair-wise test coverage is effective. In [6], Dalal et al. presented empirical results to argue that testing of all pair-wise interactions in a software system finds a large percentage of the existing faults. Kuhn et al. [7] examined fault reports for many software systems and concluded that more than 70% of the faults are triggered by two-way interaction of configuration parameters.

Covering Arrays (CAs) and Mixed Covering Arrays (MCAs) are combinatorial objects that correspond to test set in software testing. To perform effective pair-wise testing, there is a need to construct an optimal 2-way CA/MCA. The problem of finding a minimal t-covering array is NP-complete [2, 8]. Therefore, the main focus of researchers in the field of CIT is to find an effective technique to construct an optimal CA/MCA. Metaheuristic search based optimization techniques have been used by researchers to generate an (near) optimal CA/MCA. Metaheuristic techniques need longer run time than their greedy counterparts; however, greedy techniques usually need larger samples to exercise the same set of interactions [9].

In this paper we use GA to generate optimal CA/MCA for pair-wise testing. The purpose of this paper is to explore the effect of different mutation strategies on the overall performance of GA to construct (near) optimal
CAs and MCAs for pair-wise testing. The work presented here is an extension of our previous work [10] wherein the performance of GA to generate CAs/MCAs for pair-wise testing was improved by using a greedy approach to perform value occurrences mutation, a smart mutation technique introduced by Flores and Cheon [11]. Smart mutations select the genes for mutation based on some selection criteria and replace them with some predefined values as compared to random mutation in which genes for mutation are selected randomly and are replaced by randomly selected values. In this paper an algorithm to improve the performance of smart mutation and a technique to perform mutation using greedy approach is proposed.

The remainder of this paper is organized as follows. Section 2 gives the necessary background on combinatorial objects. Section 3 gives an overview of GA. Section 4 presents various methods available to construct CAs and MCAs. Section 5 describes the proposed greedy approach to improve the performance of smart mutation in GA and presents a new greedy algorithm to perform mutation. Section 6 describes the implementation of the proposed greedy approach using an open source tool P WiseGen[12] and presents experimental results to show the effectiveness of the proposed greedy approach. Section 7 concludes the paper and future plans are outlined.

II. COMBINATORIAL OBJECTS

This section gives an overview on combinatorial objects. CA and MCA are combinatorial objects with applications in several areas such as drug screening, data compression, GUI testing [13], web-application testing applications [14, 15], regression testing [16] and highly configurable system testing [17]. In CIT, a CA/MCA is constructed in such a way so as to cover each t-way combination of parameter values at least once. The effective application of CAs and MCAs in various fields has motivated researchers to find effective ways to construct optimal CA/MCA.

A. Orthogonal Arrays

An orthogonal array OAₜ(N; t, k, v) is an N × k array on v symbols such that every N × t sub-array contains all ordered subsets of size t from v symbols exactly λ times and they have the property λ = N/vᵗ [18]. The use of OA in the field of software testing is limited due to the restrictions imposed on OA that all parameters have same number of values and that each pair of values be covered the same number of times [19]. In general, the OA is difficult to generate and its test suite is often quite large. But OA has its advantages, such as making it relatively easy to identify the particular combination that caused a failure [20]. To complement OA construction and to overcome its restrictions CAs and MCAs have been introduced.

B. Covering Arrays

A covering array [21] denoted by CAₜ(N; t, k, v), is an N × k two dimensional array S on v symbols such that every N × t sub-array contains all ordered subsets from v symbols of size t at least λ times. If λ = 1, it means that every t-tuple needs to be covered only once and we can use the notation CA(N; t, k, v). N is the number of rows of S, k is the degree that represents the number of parameters and v is the order which represent the number of values each parameter can take and t is the strength that corresponds to the degree of interaction between parameters. An optimal CA contains minimum number of rows to satisfy the properties of the entire covering array. The minimum number of rows is known as covering array number and is denoted by CAN(t, k, v). A test set can be represented by a CA of size N×k where each row corresponds to a test case. Each column represents an input parameter and the values in the column represent the domain of the respective input parameter.

C. Mixed Covering Arrays

A mixed covering array [22] denoted by MCA(N; t, k, (v₁, v₂, ..., vₖ)), is an N × k two dimensional array, where v₁, v₂, ..., vₖ is a cardinality vector which indicates the values for every column. An MCA has the following two properties: 1) Each column i (1 ≤ i ≤ k) contains only elements from a set Sᵢ with |Sᵢ| = vᵢ and 2) The rows of each N × t sub-array cover all t-tuples of values from the t columns at least once. The minimum N for which there exists an MCA is called mixed covering array number and is denoted by MCAN(t, k, (v₁, v₂, ..., vₖ)). A shorthand notation can be used to represent MCAs by combining equal entries in (v₁ : 1 ≤ i ≤ k). An MCA (N; t, k, (v₁ v₂ ... vₖ)) can be represented as MCA(N; t, k, (w₁,q₁w₂,q₂ ... wₖ,qₖ)), where k = Σᵢ=₁ⁿqᵢ and wⱼ | 1 ≤ j ≤ s ⊆ {v₁ v₂ ... vₖ}. Each element wⱼ in the set (w₁,q₁w₂,q₂ ... wₖ,qₖ) means that qᵢ parameters can take wⱼ values each. A MCA of size N × k can be used to represent a test suite with N test cases for a system with k input parameters each with varying domain size. We use the notation rᵢ for all 1 ≤ i ≤ N, to represent a row of CA/MCA.

III. GENETIC ALGORITHM

GA is a metaheuristic stochastic method that is inspired by the Darwinian evolution and is used to solve search based optimization problems. GA has been successfully applied for solving large number of optimization problems due to its robustness and easy-to-use nature [23]. In GA, a population of candidate solution is initialized and evolves towards increasingly better regions of the search space by means of evolutionary operators like selection, crossover and mutation, until a satisfactory solution to the problem is found or a stopping criterion (maximum number of iterations) is met. Each individual in the population has a fitness value which is calculated using the fitness function. This fitness function is a function of the objective that we want to optimize. As compared to traditional search algorithms, GA is more flexible and can be applied to a wide range of applications as it uses only the evaluation of the objective.
function regardless of its nature. Also GA starts searching using a population of points instead of a single point (as done in case of traditional approaches) thus covering the search space thoroughly and avoids the chances of getting stuck in the local minima [24]. The efficiency of GA depends on many parameters such as initial population, selection strategy and recombination operators (crossover, mutation). The adaptation of GA parameters and operators has become an important research area in the field of GA. Over a decade much research has been done in applying adaptive mutation operators to guide the search of GA towards optimum solution.

Having described the notations, in the next section we will briefly discuss the existing state-of-the-art algorithms for constructing optimal CA/MCA for pair-wise testing.

IV. RELATED WORK

Existing state-of-the-art algorithms for constructing (near) optimal CAs and MCAs are broadly classified into algebraic, greedy, metaheuristic and random methods. Algebraic methods are generally used by mathematicians. There are two approaches to construct CAs using algebraic methods. The first approach to construct CA is based on the construction of OA, where OA is derived from some mathematical functions [19, 25]. The second approach is based on the concept of recursive construction where larger CA is constructed from smaller CA [26, 27]. Despite the fact that algebraic methods are fast, their use in the field of CIT is often limited due to the restriction imposed on OA that each parameter must have same number of values [28]. Finally, constraint handling [29] can be more difficult in algebraic methods.

Greedy algorithms have been the most popular method among software testing community to construct CAs. The algorithms used to construct CAs using greedy approach are classified as: one-test-at-a-time and one-parameter-at-a-time. In one-test-at-a-time approach, CA is constructed one row at a time and the algorithms using this approach usually differ in the way rows are constructed. The well known strategies under this approach are Automatic Efficient Test Generator (AETG) [3], Test Case Generation (TCG) [30], Classification-Tree Editor eXtended Logics (CTE-XL) [31], Jenny [32], Pairwise Independent Combinatorial Testing (PICT) [33], Deterministic Density Algorithm (DDA) [34, 35], Intersection Residual Pair Set Strategy (IRPS) [36]. In case of one-parameter-at-a-time approach, CA is generated for the first two parameters, and then it is extended to generate CA for the first three parameters, and continues to do so for each additional parameter [2, 37]. The strategies that have adapted this approach are IPOG [28], IPOG-D [28, 38] and IPO-s [39].

Recently metaheuristic techniques such as Simulated Annealing (SA) [22, 40, 41, 42, 43, 44], Hill Climbing (HC) [22], Tabu Search (TS) [45, 46, 47], Ant Colony Optimization (ACO) [48], Particle Swarm Optimization (PSO) [49, 50, 51, 52, 53, 54, 55] and GA [10, 11, 48, 56, 57, 58, 59, 60] have been explored by researchers to generate CAs/MCAs. In [61], Stadom first compared SA, TS and GA to construct CAs of strength-2 showing SA to be the most efficient of all three. Cohen et al. [22] used SA and HC to construct CAs and MCAs of strength-t (t ≤ 3) and the experimental results showed that heuristic techniques outperformed greedy methods for strength-2 CAs but they failed to give superior results for higher strength CA especially for t = 3. A comparison between SA and HC shows that while they produced similar lower bound, but SA outperformed HC in the number of trials required to generate the solution. Later on, numerous methods [40, 41, 42] have been proposed by Cohen et al. that use a combination of different methods (e.g. algebraic method and computational search) to generate uniform covering array and variable strength covering array. The existence of constraints in a system makes CIT difficult as the generated CA may contain some combination of parameter values which are invalid. Hence, careful handling of such constraints is desirable. In [62], Garvin et al. extended SA algorithm to construct CAs for constrained interaction testing. Satisfiability (SAT) solver have been used by Haich et al. [63], Yan and Zhang [64] and Banbara et al. [65] to generate t-way CA. Calvagna and Gargantini [66] use SAT modulo theory (SMT) solvers to handle constraint during the construction of CIT samples. Calvagna and Gargantini [67, 68] presented a logic based approach to generate CA for pair-wise test coverage. Finally, in case of random methods, test cases are selected randomly from the complete set of test cases based on some input distribution. They are mainly used for comparison with other test suite generation algorithms to study the effectiveness and the failure detection ability of the proposed approach [69].

V. THE PROPOSED APPROACH

The process of generating optimal CA/MCA for pair-wise testing using GA begins by creating an initial population of CAs/MCAs of size N × k randomly that represents possible solutions to the given problem. Initially N is unknown hence there are two ways to start the search process. One way is to set a loose lower and upper bounds on the size of an optimal array and then use a binary search process to find a smallest size CA/MCA [61]. Second method is to start with the size of a known CA/MCA and search for a solution. This method requires less computational resources, but the size of the CA/MCA must be known in advance. In this paper we use the second method to generate optimal CAs/MCAs, where N is chosen from the reported results (best bound achieved) in the existing state-of-the-art. After initialization, the fitness of each individual CA/MCA is evaluated using a fitness function which is defined as the number of distinct pair of parameter values covered by the CA/MCA. Then selection, crossover and mutation operators are applied iteratively to evolve the initial solution towards better solution. Mutation has a significant effect on the performance of GA as mutation avoids getting stuck in the local minima and maintains diversity in the population. In traditional GA, every
individual has an equal probability of getting mutated irrespective of their fitness [24]. Thus the probability of an individual with highest fitness to be disrupted by mutation is equal as compared to the one with lowest fitness. Hence a mutation strategy is needed to mutate an individual to maximize improvement in fitness by minimizing fitness loss due to mutation.

In this paper an effort is made:

(i) To present a greedy approach to improve the performance of pair occurrences mutation and similarity mutation [11].
(ii) To present a new greedy algorithm to perform mutation in GA for construction of optimal CA/MCA for pair-wise testing.

A. Improved_Pair Occurrences Mutation

In pair occurrences mutation [11], pairs of parameter values that are not present in the CA/MCA selected for mutation are inserted in place of pairs which occur more than once in the CA/MCA, in an attempt to increase the fitness of the CA/MCA. When an existing pair is replaced with an uncovered pair, two cases may arise:

(i) One-value replacement – In this case only one value of an existing pair needs to be replaced to accommodate the uncovered pair.
(ii) Two-value replacement – In this case both values of an existing pair needs to be replaced to accommodate the uncovered pair.

During mutation, in addition to the gain of new pairs that are formed by the insertion of an uncovered pair, there may also be a loss of few existing pairs that are formed by the combination of values of the pair which are selected for replacement. For instance, if we consider a MCA (9, 2, 5, 23459) shown in Fig. 1, it can easily be found that the pair ‘a1 a2’ is not covered by the given MCA. When examining the MCA it is found that the pair ‘a1 c2’ has maximum number of occurrences and hence the existing pair occurrences mutation replaces the first instance (row r3 in our case) of ‘a1 c2’ by ‘a1 a2’. This is the case of one-value replacement, where only the value of parameter P2 is replaced. Initially row r3 covers c2 a1, c2 a3, c2 c4 and c2 b5 pairs with respect to the parameter P2’s value c2. After replacing c2 by a2, the pairs covered by r3 with respect to a2 are a2 a1, a2 a3, a2 c4 and a2 b5. Out of these four pairs, pair a2 a3 has also been covered by row r1 of the MCA. Hence after replacement, the MCA covers three new pairs: a2 a1, a2 c4 and a2 b5. However, there is also a loss of two pairs: c2 a3 and c2 c4 since these pairs were covered by only row r3 before pair occurrences mutation was performed. The improvement in the fitness of MCA after pair occurrences mutation denoted by \( F_{\text{improved}}(\text{MCA}) \) is calculated using (1). The improvement in fitness in this case is one.

\[
F_{\text{improved}}(\text{MCA}) = \text{Number of new pairs gained} - \text{Number of old pairs lost}
\]  

As an existing pair is replaced by a missing pair in either one-value replacement or two-value replacement, there are three possible cases that can occur during replacement: best, worst and average case. In the best case we assume that there will be a gain of maximum possible number of new pairs (k-1) in one-value replacement and (2k-3) in two-value replacement whereas the loss will be minimum (ideally zero) in both the cases. In the worst case we assume that in both the cases, there will be a gain of only one pair which was not covered by the MCA before mutation. However, a maximum loss of (k-2) pairs in case of one-value replacement and (2k-4) pairs in case of two-value replacement will occur. So the improvement in fitness in all the three cases for one-value replacement and two-value replacement is given by (2) and (3) respectively:

\[
F_{\text{improved}}(\text{MCA}) = \left\{ \begin{array}{ll}
(k - 1) & \text{best case} \\
-(k - 3) & \text{worst case} \\
-(k - 3) < x < (k - 1) & \text{average case}
\end{array} \right.
\]  

\[
F_{\text{improved}}(\text{MCA}) = \left\{ \begin{array}{ll}
-(2k - 3) & \text{best case} \\
-(2k - 5) & \text{worst case} \\
-(2k - 5) < x < (2k - 3) & \text{average case}
\end{array} \right.
\]
the value that is to be replaced. Then the listed pairs of each candidate row are compared against each other and if a row is found that covers no distinct pairs then it is selected for replacement. If no such row is found, the listed pairs of each candidate row are compared with the remaining rows of the MCA and the candidate row \( \mathbf{r}_1 \) covering least number of distinct pairs is selected for replacing value during pair-occurrences mutation. In two-value-replacement the same procedure is repeated, except the way the pairs are listed. Now we list all the pairs covered by both values of the pair which occurs maximum number of times and select the row which covers least number of distinct pairs. When we apply improved_pair_occurrences mutation in the example given in Fig. 1, it is found that row \( \mathbf{r}_6 \) covers no distinct pairs as shown in Fig. 2, so we select row \( \mathbf{r}_6 \) instead of row \( \mathbf{r}_3 \) to minimize the loss of existing pairs and thus maximize the fitness of the MCA by four.

\[
F_{\text{improved}}(\text{MCA}) = \begin{cases} 
  k(k-1)/2 & \text{best case} \\
  -(2kn-n^2-n)/2 & \text{worst case} \\
  -(2kn-n^2-n)/2 < x < k(k-1)/2 & \text{average case} 
\end{cases}
\]

(4)

Where, \( n \) is the numbers of positions in which the two rows differ.

In this paper, we propose a technique to maximize \( F_{\text{improved}}(\text{MCA}) \) by minimizing the loss of existing pairs during similarity mutation. A greedy approach is used to select from the two or more similar rows, a row which covers least number of distinct pairs within the given MCA. An example is shown in Fig. 3 to illustrate the effect of similarity mutation and improved_similarity mutation. Test cases represented by row \( \mathbf{r}_2 \) and \( \mathbf{r}_4 \) of MCA \((10, 2, 6, 4^33^52^1)\) in Fig. 3 are similar (for similarity threshold=65%). In similarity mutation, \( \mathbf{r}_2 \) will get replaced with a new test case that causes a loss of five existing pairs which were not covered by any other row of the given MCA. When we apply improved_similarity mutation, row \( \mathbf{r}_2 \) and \( \mathbf{r}_4 \) are compared with every row of the given MCA and it is found that row \( \mathbf{r}_2 \) covers less number of distinct pairs as compared to row \( \mathbf{r}_4 \), hence row \( \mathbf{r}_2 \) is replaced instead of row \( \mathbf{r}_4 \), resulting in loss of less number of distinct pairs as compared to those in similarity mutation. It is clear from Fig. 3 that the improvement in fitness achieved by improved_similarity mutation will be higher than that of similarity mutation.

**B. Improved_Similarity Mutation**

In case of similarity mutation proposed by Flores and Cheons [11], if two test cases (rows) in the test set (CA/MCA) are (almost) similar (greater than or equal to a predefined threshold), the second test case is replaced with a new test case with an aim of improving the fitness of the test set. The parameter values for the new test case can be selected either randomly or for each parameter the value which occurs minimum number of times in the CA/MCA is selected. If the two test cases are 100% similar, then there will not be any loss of existing pairs during replacement otherwise there are chances that some pairs that were exclusively covered by the second test case might get lost during replacement. As explained in case of pair occurrences mutation, the improvement in fitness in best, worst and average case during similarity mutation is given by (4).
### C. Minimum Distinct Pairs Mutation (MDPM)

**Algorithm: Minimum Distinct Pairs Mutation (MDPM)**

**begin**
select a MCA for mutation  
set flag = true  
while (flag = = true)  
set flag = false  
for each row \( r_i \) of MCA  
list the pairs covered by row \( r_i \)  
end for  
compare the pairs covered by each row  
if exist(\( r_i \) covers minimum distinct pairs && (number of distinct pairs covered < MDPM threshold))  
set flag = true  
replace \( r_i \) in MCA with a new row  
end if  
end while  
**end**

(Fig. 4. Algorithm to perform MDPM)

In this section we propose a greedy technique to perform mutation known as minimum distinct pairs mutation (MDPM). As described above, in case of similarity mutation we replace a similar test case, but it may happen that there are no test cases (rows) which satisfy the similarity threshold criteria. In that case similarity mutation cannot be performed. However, there may be a test case which is not similar to any other test case in the given MCA, but still makes no contribution towards the fitness of the MCA. In MDPM we replace the test case which makes minimum contribution (ideally zero) towards the fitness of MCA by covering least number of distinct pairs as compared to the remaining test cases in the MCA. A MDPM threshold needs to be defined here, which allows MDPM to occur only if the number of distinct pairs covered by the test case is below the MDPM threshold value. The MDPM threshold prevents good test cases to be unnecessarily distorted by mutation. During MDPM, the parameter values for the new test case can be selected either randomly or for each parameter the value which occurs minimum number of times in the CA/MCA is selected as in case of similarity mutation. For instance, in the example shown in Fig. 3, row \( r_1 \) is not similar to any other row of the MCA (for similarity threshold = 65%). However when we list all pairs covered by \( r_1 \) and compare them with pairs covered by the remaining (N-1) rows of the MCA, it has been observed that row \( r_1 \) doesn’t cover any distinct pair with respect to the remaining (N-1) rows. Hence, we replace \( r_1 \) by a new test case. The objective of MDPM is to maximize the improvement in fitness of MCA after mutation by minimizing the losses. It can be seen from Fig. 3 that row \( r_1 \) gets replaced after similarity mutation causing a loss of five pairs, row \( r_2 \) gets replaced after improved_similarity mutation reducing the loss to two pairs whereas row \( r_7 \) is replaced after MDPM dropping the loss to zero pair. One point that is to be noted here is that the performance of MDPM is comparable to similarity/improved_similarity mutation if two rows in the MCA are 100% similar. An algorithm to perform MDPM is given in Fig. 4.

The improvement in fitness during MDPM in best, worst and average case is given by (5).

\[
F_{\text{improved}} (\text{MCA}) = \begin{cases} 
\frac{k(k-1)}{2} & \text{best case} \\
0 & \text{worst case} \\
0 < x < \frac{k(k-1)}{2} & \text{average case} 
\end{cases}
\]

(5)

The advantage of MDPM over similarity and improved_similarity mutation is its performance in average and worst case. It is observed from (5) that in average and worst case, the quality of MCA generated after MDPM doesn’t deteriorate as may happen in case of similarity and improved_similarity mutation.

**VI. COMPUTATIONAL RESULTS**

To assess the practicality of the work presented in this paper, we have implemented the proposed approaches using an open source tool PWiseGen. PWiseGen is an extensible, reusable and configurable tool written in Java to generate pair-wise test set using GA [11]. We have extended PWiseGen by adding to it the capability to perform improved_pair occurrences mutation, improved_similarity mutation and MDPM and name it PWiseGen-GM (Greedy Mutation). First, we present the results of experiments carried out to compare the performance of existing smart mutations with the proposed improved smart mutations. Next, we compare the performances of improved smart mutations with MDPM. Finally we compare the performance of PWiseGen with MDPM with the existing state-of-the-art algorithms.

Experiments are carried out on the dataset given in Table 2. The dataset consists of benchmark problems selected from the existing state-of-the-art [22, 43, 52, 54, 70] for generating both CAs and MCAs.

For achieving the best performance of PWiseGen-GM it is necessary to choose suitable values of GA parameters. There exists evidence in literature that the choice of probability of crossover \( p_c \) and probability of mutation \( p_m \) plays a critical role in the performance of GA. A number of guidelines for setting the values of \( p_c \) and \( p_m \) exist in literature [23, 71, 72]. Typical values of \( p_c \) are in the range 0.5–1.0, while typical values of \( p_m \) are in the range 0.001–0.05. In addition to the parameters mentioned above, population size and number of generations influences the performance of GA. A large population size will use more computational resources without obtaining better solutions, whereas a small population size may lead to the under-covering of search space thereby guiding the algorithm towards poor solutions. Similarly a large number of generations may consume more time, whereas a smaller number of generations will make the algorithm terminate early, preventing it to
converge to a good solution. Hence in accordance with the existing GA literature, we set the population size to 50, $p_c$ to 1 and $p_m$ to 0.05 during the experimentation. We perform experiments with varying number of generations on some selected benchmark problems from Table 2, but no significant improvements on quality of the final solution were observed after 20,000 generations in all the cases. Hence we set the number of maximum generations to 20,000.

Table 1. Dataset

<table>
<thead>
<tr>
<th>Sno.</th>
<th>Benchmark Problems</th>
<th>$k$ (number of input parameters)</th>
<th>Total number of pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$3^2$</td>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>$3^2$</td>
<td>4</td>
<td>54</td>
</tr>
<tr>
<td>3</td>
<td>$3^2$</td>
<td>5</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>$3^6$</td>
<td>6</td>
<td>135</td>
</tr>
<tr>
<td>5</td>
<td>$3^7$</td>
<td>7</td>
<td>189</td>
</tr>
<tr>
<td>6</td>
<td>$3^8$</td>
<td>8</td>
<td>252</td>
</tr>
<tr>
<td>7</td>
<td>$3^{13}$</td>
<td>13</td>
<td>702</td>
</tr>
<tr>
<td>8</td>
<td>$5^{10}$</td>
<td>10</td>
<td>1125</td>
</tr>
<tr>
<td>9</td>
<td>$10^{29}$</td>
<td>20</td>
<td>19000</td>
</tr>
<tr>
<td>10</td>
<td>$2^{100}$</td>
<td>100</td>
<td>19800</td>
</tr>
<tr>
<td>11</td>
<td>$4^{100}$</td>
<td>100</td>
<td>79200</td>
</tr>
<tr>
<td>12</td>
<td>$2^{33}$</td>
<td>5</td>
<td>67</td>
</tr>
<tr>
<td>13</td>
<td>$4^{39}$</td>
<td>9</td>
<td>454</td>
</tr>
<tr>
<td>14</td>
<td>$5^{43^2}$</td>
<td>11</td>
<td>492</td>
</tr>
<tr>
<td>15</td>
<td>$2^3\times4^1\times6^2$</td>
<td>12</td>
<td>837</td>
</tr>
<tr>
<td>16</td>
<td>$7^6\times3^2\times2^5$</td>
<td>10</td>
<td>854</td>
</tr>
<tr>
<td>17</td>
<td>$2^{12}$</td>
<td>18</td>
<td>992</td>
</tr>
<tr>
<td>18</td>
<td>$8^{7}\times6^5$</td>
<td>8</td>
<td>1178</td>
</tr>
<tr>
<td>19</td>
<td>$6^4\times2^6$</td>
<td>16</td>
<td>1556</td>
</tr>
<tr>
<td>20</td>
<td>$5^4\times3^{11}5^2$</td>
<td>21</td>
<td>1944</td>
</tr>
<tr>
<td>21</td>
<td>$6^5\times3^{3}3^2$</td>
<td>19</td>
<td>1992</td>
</tr>
<tr>
<td>22</td>
<td>$6^4\times2^5$</td>
<td>20</td>
<td>2052</td>
</tr>
<tr>
<td>23</td>
<td>$6^5\times3^4$</td>
<td>14</td>
<td>2074</td>
</tr>
<tr>
<td>24</td>
<td>$7^6\times5^4\times3^2$</td>
<td>19</td>
<td>2175</td>
</tr>
<tr>
<td>25</td>
<td>$6^4\times2^7$</td>
<td>19</td>
<td>3000</td>
</tr>
<tr>
<td>26</td>
<td>$6^4\times2^5$</td>
<td>18</td>
<td>3004</td>
</tr>
<tr>
<td>27</td>
<td>$4^3\times7^2\times9^3\times2^5$</td>
<td>61</td>
<td>14026</td>
</tr>
<tr>
<td>28</td>
<td>$4^3\times3^2\times2^5$</td>
<td>75</td>
<td>17987</td>
</tr>
</tbody>
</table>

A. Improved Smart Mutations versus Smart Mutations

In this section we present the result of experiments carried out to compare the performance of a) improved_pair occurrences mutation and pair occurrences mutation and b) improved_similarity mutation and similarity mutation. In both the cases, first we plotted the fitness of the generated CA/MCA against the generation number. Second, we made a comparison between the fitness of best CA/MCA generated by the existing smart mutation techniques and the proposed improved smart mutation techniques. As GA produces non-deterministic results, so to have a better statistical significance, each benchmark problem is executed 30 times and the average value is noted. The result of comparison of improved_pair occurrences mutation and pair occurrences mutation is shown in Fig. 5 and Fig. 6. Due to space reason, we show the experimental results of a few benchmark problems selected as representative, from the dataset of Table 1.
It is clear from Fig. 5 that improved_pair occurrences mutation generates CA/MCA in less number of generations than pair occurrences mutation. Also, it is evident from Fig. 6 that the quality of CA/MCA generated using improved_pair occurrences mutation is better than that of pair occurrences mutation. The result of comparison of improved_similarity mutation and similarity mutation is shown in Fig. 7 and Fig. 8.

It is evident from Fig. 7 and Fig. 8 that the CA/MCAs generated using improved_similarity mutation generates better quality CA/MCA by covering more number of distinct pairs in less number of generations as compared to similarity mutation.

B. Comparison of MDPM and Improved Smart Mutations

In this section, we present the result of experiments carried out to evaluate the performance of MDPM with respect to the improved smart mutations techniques proposed in Section 5 and in [10]. Each benchmark problem given in the dataset of Table 2 is executed 30 times on PWiseGen-GM using various improved smart mutations techniques and MDPM. The average of the values obtained over 30 runs for each of the benchmark problem is plotted as shown in Fig. 9. It is evident from Fig. 9 that MDPM outperforms improved smart mutation techniques by generating better quality CA/MCA. The performance of improved_value occurrences mutation is comparable to the performance of improved_pair occurrences mutation except in few cases where improved_pair occurrences mutation outperforms improved_value occurrences mutation. For small size problems, the performances of all the four mutation strategies are identical (benchmark problems: 3^3 and 3^4).

C. Comparison Results

In this Section we compare the performance PWiseGen-GM with MDPM with the existing state-of-the-art algorithms. The comparison is made on two criteria’s: array size and array generation time. As array generation time is dependent on the system configuration, so to ensure a fair comparison, we restrict our comparison...
against publicly available tools namely Jenny [32], Pairwise Independent Combinatorial Testing PICT [33], ACTS (IPOG) [73], AllPairs [74]. These algorithms are run on Windows using an INTEL Pentium Dual Core 1.73 GHZ processor with 3.00 GB of memory. The result of comparison made on the dataset of Table 2 with respect to CA/MCA array size and the array generation time (in seconds) is shown in Table 3.

As it can be seen from Table 3, PWiseGen-GM with MDPM outperforms the existing techniques for generating CA/MCA for pair-wise testing in most of the configurations. In cases where PWiseGen-GM (MDPM) doesn’t generate best results, it still gives competitive performances than the existing strategies. It is evident from the results shown in Table 3 that PWiseGen-GM (MDPM) requires more time from the results shown in Table 3 that PWiseGen-GM performs better than the existing strategies. It is evident from the results shown in Table 3 that PWiseGen-GM (MDPM) doesn’t generate best results, it still gives competitive performances than the existing strategies. It is evident from the results shown in Table 3 that PWiseGen-GM (MDPM) doesn’t generate best results, it still gives competitive performances than the existing strategies. It is evident from the results shown in Table 3 that PWiseGen-GM (MDPM) doesn’t generate best results, it still gives competitive performances than the existing strategies. It is evident from the results shown in Table 3 that PWiseGen-GM (MDPM) doesn’t generate best results, it still gives competitive performances than the existing strategies. It is evident from the results shown in Table 3 that PWiseGen-GM (MDPM) doesn’t generate best results, it still gives competitive performances than the existing strategies. It is evident from the results shown in Table 3 that PWiseGen-GM (MDPM) doesn’t generate best results, it still gives competitive performances than the existing strategies. It is evident from the results shown in Table 3 that PWiseGen-GM (MDPM) doesn’t generate best results, it still gives competitive performances than the existing strategies. It is evident from the results shown in Table 3 that PWiseGen-GM (MDPM) doesn’t generate best results, it still gives competitive performances than the existing strategies.


[70] http://www.pairwise.org


[73] ACTS download page, National Institute of Standards and Technology, Information Technology Laboratory.


Authors’ Profiles

Sangeeta Sabharwal did her M.Tech in Computer Science and Ph.D from University of Delhi, India. Presently she is a Professor and Head, Division of Information Technology at NSIT, University of Delhi, India. She has around 25 years of experience in the field of software engineering. Her areas of interest are model based testing, web application testing, search based software engineering and meta modeling.

Priti Bansal received her B.E in Computer Science from University of Mumbai, India in 2001 and her M.Tech in Information System from University of Delhi, India in 2009. She is pursuing her PhD and currently working as Assistant Professor in the Division of Information Technology, NSIT, New Delhi, India. Her research interest includes model based testing, web application testing, search based software engineering and neural networks.

Nitish Mittal is pursuing Bachelor of Engineering in Division of Computer Engineering from Netaji Subhas Institute of Technology, New Delhi, India. His areas of interest are software testing, soft computing and data mining.